Interval Slopes as a Numerical Abstract Domain for Floating-Point Variables

Alexandre Chapoutot
LIP6 - Université Pierre et Marie Curie
4, place Jussieu F-75252 Paris Cedex 05 France
alexandre.chapoutot@lip6.fr

June 21, 2010

Abstract

The design of embedded control systems is mainly done with model-based tools such as Matlab/Simulink. Numerical simulation is the central technique of development and verification of such tools. Floating-point arithmetic, which is well-known to only provide approximated results, is omnipresent in this activity. In order to validate the behaviors of numerical simulations using abstract interpretation-based static analysis, we present, theoretically and with experiments, a new partially relational abstract domain dedicated to floating-point variables. It comes from interval expansion of non-linear functions using slopes and it is able to mimic all the behaviors of the floating-point arithmetic. Hence it is adapted to prove the absence of run-time errors or to analyze the numerical precision of embedded control systems.

1 Introduction

Embedded control systems are made of a software and a physical environment which aim at continuously interact with each other. The design of such systems is usually realized with the model-based paradigm. Matlab/Simulink[1] is one of the most used tools for this purpose. It offers a convenient way to describe the software and the physical environment in an unified formalism. In order to verify that the control law, implemented in the software, fits the specification of the system, several numerical simulations are made under Matlab/Simulink. Nevertheless, this method is closer to test-based method than formal proof. Moreover, this verification method is strongly related to the floating-point arithmetic which provides approximated results.

Our goal is the use of abstract interpretation-based static analysis [9] to validate the design of control embedded software described in Matlab/Simulink. In our previous work [3], we defined an analysis to validate that the behaviors given by numerical simulations are close to the exact mathematical behaviors. It was based on an interval abstraction of floating-point numbers which may produce too coarse results. In this article, our work is focused on a tight representation of the behaviors of the floating-point arithmetic in order to increase the precision of the analysis of Matlab/Simulink models.

To emphasize the poor mathematical properties of the floating-point arithmetic, let us consider the sum of numbers given in Example 1 with a single precision floating-point arithmetic. The result of this sum is $-2.08616257.10^{-6}$ due to rounding errors, whereas the exact mathematical result is zero.

Example 1.

$$\begin{aligned}0.0007 + (-0.0097) + 0.0738 + (-0.3122) + 0.7102 + (-0.5709) + (-1.0953)\\ + 3.3002 + (-2.9619) + (-0.2353) + 2.4214 + (-1.7331) + 0.4121\end{aligned}$$

[1]Trademarks of The Mathworks^{TM}company.

Example 1 shows that the summation of floating-point numbers is a very ill-conditioned problem [28, Chap. 6]. Indeed, small perturbations on the elements to sum produce a floating-point result which could be far from the exact result. Nevertheless, it is a very common operation in control embedded software. In particular, it is used in filtering algorithms or in regulation processes, such as for example in PID[2] regulation. Remark that depending on the case, the rounding errors may stay insignificant and the behaviors of floating-point arithmetic may be safe. In consequence, a semantic model of this arithmetic could be used to prove the behaviors of embedded control software using floating-point numbers.

The definition of abstract numerical domains for floating-point numbers is usually based on rational or real numbers [13, 24] to cope with the poor mathematical structure of the floating-point set. In consequence, these domains give an over-approximation of the floating-point behaviors. This is because they do not bring information about the kind of numerical instability appearing during computations. We underline that our goal is not interested in computing the rounding errors but the floating-point result. In others words, we want to compute the bounds of floating-point variables without considering the numerical quality of these bounds.

Our main contribution is the definition of a new numerical abstract domain, called Floating-Point Slopes (FPS), dedicated to the study of floating-point numbers. It is based on interval expansion of non-linear functions named *interval slopes* introduced by Krawczyk and Neumaier [20] and, as we will show in this article, it is a partially relational domain. The main difference is that, in Proposition 1, we adapt the interval slopes to deal with floating-point numbers. Moreover, we are able to tightly represent the behaviors of floating-point arithmetic with our domain. A few cases studies will show the practical use of our domain. Hence we can prove properties on programs taking into account the behaviors of the floating-point arithmetic such that the absence of run-time errors or, by combining it with other domains *e.g.* [4], the quality of numerical computations.

Content. In Section 2, we will present the main features of floating-point arithmetic and we will also introduce the interval expansions of functions. We will present our abstract domain FPS in Section 3 and the analysis of floating-point programs in Section 4 before describing experimental results in Section 5. In Section 6, we will reference the related work before concluding in Section 7.

2 Background

We recall the main features of the IEEE754-2008 standard of floating-point arithmetic in Section 2.1. Next in Section 2.2, we present some results from interval analysis, in particular the interval expansion of functions.

2.1 Floating-Point Arithmetic

We briefly present the floating-point arithmetic, more details are available in [28] and the references therein. The IEEE754-2008 standard [18] defines the floating-point arithmetic in base 2 which is used in almost every computer[3].

Floating-point numbers have the following form: $f = s.m.2^e$. The value s represents the *sign*, the value m is the *significand* represented with p bits and the value e is the *exponent* of the floating-point number f which belongs into the interval $[e_{\min}, e_{\max}]$ such that $e_{\max} = -e_{\min} + 1$. There are two kinds of numbers in this representation. *Normalize numbers* for which the significand implicitly starts with a 1 and *denormalized numbers* that implicitly starts with a 0. The later are used to gain accuracy around zero by slowly degrading the precision.

The standard defines different values of p and $e_{\min}$: $p = 24$ and $e_{\min} = -126$ for the single precision and $p = 53$ and $e_{\min} = -1022$ for the double precision. We call *normal range* the set of absolute real values in $[2^{e_{\min}}, (2 - 2^{1-p})2^{e_{\max}}]$ and the *subnormal range* the set of numbers in $[0, 2^{e_{\min}}[$.

[2]PID stands for proportional-integral-derivative. It is a generic method of feedback loop control widely used in industry.
[3]It also defines this arithmetic in base 10 but it is not relevant for our purpose.

The set of floating-point numbers (single or double precision) is represented by $\mathbb{F}$ which is closed under negation. A few special values represent special cases: the values $-\infty$ and $+\infty$ to represent the negative or the positive overflow; and the value NaN[4] represents invalid results such that $\sqrt{-1}$.

The standard defines round-off functions which convert exact real numbers into floating-point numbers. We are mainly concerned by the rounding to the nearest ties to even[5] (noted fl), the rounding towards $+\infty$ and rounding toward $-\infty$. The round-off functions follow the correct rounding property, *i.e.* the result of a floating-point operation is the same that the rounding of the exact mathematical result. Note that these functions are monotone. We are interested in this article by computing the range of floating-point variables rounded to the nearest which is the default mode of rounding in computers.

A property of the round-off function fl is given in Equation (1). It characterizes the *overflow*, *i.e.* the rounding result is greater than the biggest element of $\mathbb{F}$ and the case of the generation of 0. This definition only uses positive numbers, using the symmetry property of $\mathbb{F}$, we can easily deduce the definition for the negative part. We denote by $\sigma = 2^{e_{\min}-p+1}$ the smallest positive subnormal number and the largest finite floating-point number by $\Sigma = (2 - 2^{1-p})2^{e_{\max}}$.

$$\forall x \in \mathbb{F}, x > 0, \quad \mathrm{fl}(x) = \begin{cases} +0 & \text{if } 0 < x \leq \sigma/2 \\ +\infty & \text{if } x \geq \Sigma \end{cases} \tag{1}$$

An *underflow* [28, Sect. 2.3] is detected when the rounding result is less than $2^{e_{\min}}$, *i.e.* the result is in the subnormal range.

The errors associated to a correct rounding is defined in Equation (2) and it is valid for all floating-point numbers x and y except $-\infty$ and $+\infty$ (see [28, Chap. 2, Sect. 2.2]). The operation $\diamond \in \{+, -, \times, \div\}$ but it is also valid for the square root. The *relative rounding error unit* is denoted by μ. In single precision, $\mu = 2^{-24}$ and $\sigma = 2^{-149}$ and in double precision, $\mu = 2^{-53}$ and $\sigma = 2^{-1074}$.

$$\mathtt{fl}(x \diamond y) = (x \diamond y)(1 + \epsilon_1) + \epsilon_2 \qquad \text{with } |\epsilon_1| \leq \mu \text{ and } |\epsilon_2| \leq \frac{1}{2}\sigma \tag{2}$$

If $\mathtt{fl}(x \diamond y)$ is in the normal range or if $\diamond \in \{+, -\}$ then ϵ_2 is equal to zero. If $\mathtt{fl}(x \diamond y)$ is in the subnormal range then ϵ_1 is equal to zero.

Numerical instabilities in programs come from the rounding representation of values and they also came from two problems due to finite precision:

Absorption If $|x| \leq \mu|y|$ then it happens that $\mathrm{fl}(x + y) = \mathrm{fl}(y)$. For example, in single precision, the result of $\mathrm{fl}(10^4 - 10^{-4})$ is $\mathrm{fl}(10^4)$. In numerical analysis, the solution avoid this phenomenon is to sort the sequence of numbers [17, Chap. 4]. This solution is not applicable when the numbers to add are given by a sensor measuring the physical environment.

Cancellation It appears in the subtraction $\mathrm{fl}(x - y)$ if $(|x - y|) \leq \mu(|x| + |y|)$ then the relative errors can be arbitrary big. Indeed, the rounding errors take usually place in the least significant digits of floating-point numbers. These errors may become preponderant in the result of a subtraction when the most significant digits of two closed numbers cancelled each others. In numerical analysis, subtraction of numbers coming from long computations are avoided to limit this phenomena. We cannot apply this solution in embedded control systems where some results are used at different instants of time.

2.2 Interval Arithmetic

We introduce interval arithmetic and in particular, the interval expansion of functions which is an element of our abstract domain FPS.

[4] *NaN* stands for *Not A Number.*

[5] The IEEE754-2008 standard introduces two rounding modes to the nearest with respect to the previous IEEE754-1985 and IEEE754-1987 standards. These two modes only differ when an exact result is in half-way of two floating-point numbers. In rounding-to-nearest-tie-to-even mode, the floating-point number whose the least significand bit is even is chosen. Note that this definition is used in all the other revisions of the IEEE754 standard, see [28, Chap. 3.4] for more details.

2.2.1 Standard Interval Arithmetic.

The *interval arithmetic* [27] has been defined to avoid the problem of approximated results coming from the floating-point arithmetic. It had also been used as the first numerical abstract domain in [9].

When dealing with floating-point intervals the bounds have to be rounded to outward as in [24, Sect. 3]. In Example 2, we give the result of the interval evaluation in single precision of a sum of floating-point numbers.

Example 2. *Using the interval domain for floating-point arithmetic [24, Sect. 3] the result of the sum defined by* $\sum_{i=1}^{10} 10^1 + \sum_{i=1}^{10} 10^2 + \sum_{i=1}^{10} 10^3 + \sum_{i=1}^{1000} 10^{-3}$ *is* $[11100, 11101.953]$. *The exact result is* 11101 *while the floating-point result is* 11100 *due to an absorption phenomena. The floating-point result and the exact result are in the result interval but we cannot distinguish them any more.*

A source of over-approximation is known in the interval arithmetic as the *dependency problem* which is also known in static analysis as the non-relational aspect. For example, if some variable has value $[a, b]$, then the result of $x - x$ is $[a - b, b - a]$ which is equal to zero only if $a = b$. This problem is addressed by considering interval expansions of functions.

Notations. We denote by x a real number and by $\vec{x}$ a vector of real numbers. Interval values are in capital letters X or denoted by $[a, b]$ where a is the lower bound and b is the upper bound of the interval. A vector of interval values will be denoted by $\vec{\mathrm{X}}$. We denote by $[\mathsf{f}]$ the interval extension of a function f obtained by substitution of all the arithmetic operations with their equivalent in interval. The center of an interval $[a, b]$ is represented by $\mathsf{mid}([a, b]) = a + 0.5 \times (b - a)$.

2.2.2 Extended Interval Arithmetic.

We are interested in the computation of the image of a vector of interval $\vec{\mathrm{X}}$ by a non-linear function $\mathsf{f} : \mathbb{R}^n \to \mathbb{R}$ only composed by additions, subtractions, multiplications and divisions and square root. In order to reduce over-approximations in the interval arithmetic, some interval expansions have been developed. The first one is based on the *Mean-Value Theorem* and it is expressed as:

$$\mathsf{f}(\vec{\mathrm{X}}) \subseteq \mathsf{f}(\vec{z}) + [\mathsf{f}'](\vec{\mathrm{X}})(\vec{\mathrm{X}} - \vec{z}) \quad \forall \vec{z} \in \vec{\mathrm{X}} \ . \tag{3}$$

The first-order approximation of the range of a function f can be defined thanks to its first order derivative f' over $\vec{\mathrm{X}}$. We can then approximate $\mathsf{f}(\vec{\mathrm{X}})$ by a pair $(\mathsf{f}(\vec{z}), [\mathsf{f}'](\vec{\mathrm{X}}))$ that are the value of f at point z and the interval extension of f' evaluated over $\vec{\mathrm{X}}$.

A second interval expansion has been defined by Krawczyk and Neumaier [20] using the notion of slopes which reduced the approximation of the derivative form. It is defined by the relation:

$$\begin{aligned} &\mathsf{f}(\vec{\mathrm{X}}) \subseteq \mathsf{f}(\vec{z}) + [\mathtt{F}^{\vec{z}}](\vec{\mathrm{X}})(\vec{\mathrm{X}} - \vec{z}) \\ &\qquad\qquad \text{with } \mathtt{F}^{\vec{z}}(\vec{\mathrm{X}}) = \left\{ \frac{\mathsf{f}(\vec{x}) - \mathsf{f}(\vec{z})}{\vec{x} - \vec{z}} : \vec{x} \in \vec{\mathrm{X}} \wedge \vec{z} \neq \vec{x} \right\} \ . \end{aligned} \tag{4}$$

Then we can represent $\mathsf{f}(\vec{\mathrm{X}})$ by a pair $(\mathsf{f}(\vec{z}), [\mathtt{F}^{\vec{z}}](\vec{\mathrm{X}}))$ that are the value of f in the point $\vec{z}$ and the interval extension of the slope $\mathtt{F}^z(X)$ of f.

Note that the value $\vec{z}$ is constructed, in general, from the centers of the interval variables appearing in the function f for both interval expansions.

An interesting feature is that we can inductively compute the derivative or the slope of a functions using *automatic differentiation* techniques [1]. It is a semantic-based method to compute derivatives. In this context, we call *independent variables* some input variables of a program with respect to which derivatives are computed. We call *dependent variables* output variables whose derivatives are desired. A *derivative object* represents derivative information, such as a vector of partial derivatives like $(\partial e/\partial x_1, \ldots, \partial e/\partial x_n)$ of some expression e with respect to a vector $\vec{x}$ of independent variables. The main idea of automatic differentiation is that every complicated function f, *i.e.* a program, is composed by simplest elements, *i.e.* program instructions.

Knowing the derivatives of these elements with respect to some independent variables, we can compute the derivatives or the slopes of f following the differential calculus rules. Furthermore, using interval arithmetic in the differential calculus rules, we can guarantee the result.

We give in Table 1 the rules to compute derivatives or slopes with respect to the structure of arithmetic expressions. We assume that we know the number of independent variables in the programs and we denote by n this number. The variable $\mathcal{V}^{\text{ind}}$ represents the vector of independent variables with respect to which the derivatives are computed. We denote by δ_i the interval vector of length n, having all its coordinates equal to $[0,0]$ except the i-th element equals to $[1,1]$. So, we consider that all the independent variables are assigned to a unique position i in $\mathcal{V}^{\text{ind}}$ and it is initially assigned with a derivative object equal to δ_i. Following Table 1 where g and h represent variables with derivative object, a constant value c has a derivative object equal to zero (the interval vector $\vec{0}$ has all its coordinates equal to $[0,0]$). For addition and subtraction, the result is the vector addition or the vector subtraction of the derivative objects. For multiplication and division, it is more complicated but the rules come from the standard rules of the composition of derivatives, *e.g.* $(u \times v)' = u' \times v + u \times v'$. A proof of the computation rules[6] for slopes can be found in [30, Sect. 1]. Note that we can apply automatic differentiation for other functions, such as the square root, using the rule of function composition, $(f \circ g)'(x) = f'(g(x))g'(x)$.

These interval expansions of functions, using either $(\mathsf{f}(\vec{z}), [\mathsf{f}'](\vec{\mathrm{X}}))$ the derivative form or $(\mathsf{f}(\vec{z}), [\mathtt{f}^{\vec{z}}](\vec{\mathrm{X}}))$ the slope form, define a straightforward semantics of arithmetic expressions which can be used to compute bounds of variables.

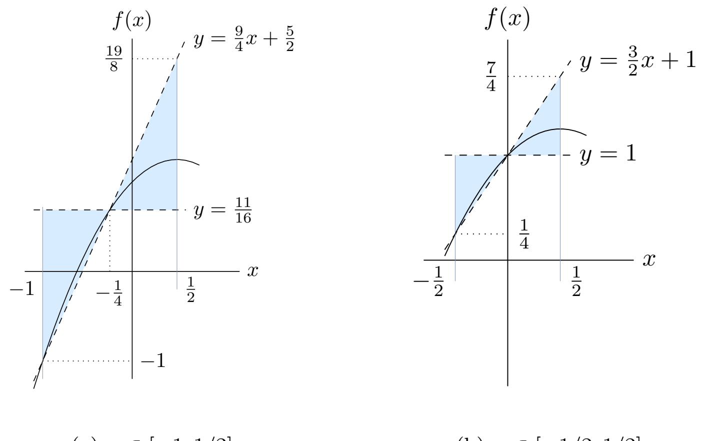
Table 1: Automatic differentiation rules for derivatives and slopes

Function	Derivative arithmetic	Slope arithmetic
$c \in \mathbb{R}$	$\mathbf{0}$	$\mathbf{0}$
$\mathsf{g}+\mathsf{h}$	$[\mathsf{g}'](\vec{\mathrm{X}}) + [\mathsf{h}'](\vec{\mathrm{X}})$	$[\mathtt{G}^{\vec{z}}](\vec{\mathrm{X}}) + [\mathtt{H}^{\vec{z}}](\vec{\mathrm{X}})$
$\mathsf{g}-\mathsf{h}$	$[\mathsf{g}'](\vec{\mathrm{X}}) - [\mathsf{h}'](\vec{\mathrm{X}})$	$[\mathtt{G}^{\vec{z}}](\vec{\mathrm{X}}) - [\mathtt{H}^{\vec{z}}](\vec{\mathrm{X}})$
$\mathsf{g}\times\mathsf{h}$	$[\mathsf{g}'](\vec{\mathrm{X}}) \times \mathsf{h}(\vec{\mathrm{X}}) + \mathsf{g}(\vec{\mathrm{X}}) \times [\mathsf{h}'](\vec{\mathrm{X}})$	$[\mathtt{G}^{\vec{z}}](\vec{\mathrm{X}}) \times \mathsf{h}(\vec{\mathrm{X}}) + \mathsf{g}(\vec{z}) \times [\mathtt{H}^{\vec{z}}](\vec{\mathrm{X}})$
$\dfrac{\mathsf{g}}{\mathsf{h}}$	$\dfrac{[\mathsf{g}'](\vec{\mathrm{X}}) \times \mathsf{h}(\vec{\mathrm{X}}) - [\mathsf{h}'](\vec{\mathrm{X}}) \times \mathsf{g}(\vec{\mathrm{X}})}{\mathsf{h}^2(\vec{\mathrm{X}})}$	$\dfrac{[\mathtt{G}^{\vec{z}}](\vec{\mathrm{X}}) - [\mathtt{H}^{\vec{z}}](\vec{\mathrm{X}}) \times \frac{\mathsf{g}(\vec{z})}{\mathsf{h}(\vec{z})}}{\mathsf{h}(\vec{\mathrm{X}})}$
$\sqrt{\mathsf{g}}$	$\dfrac{1}{2}\dfrac{[\mathsf{g}'](\vec{\mathrm{X}})}{\sqrt{\mathsf{g}(\vec{\mathrm{X}})}}$	$\dfrac{[\mathtt{G}^{\vec{z}}](\vec{\mathrm{X}})}{\sqrt{g(\vec{z})} + \sqrt{g(\vec{\mathrm{X}})}}$

Remark 1. *The difference in over-approximated result between the derivative form and the slope form is in the multiplication and the division rules. In the derivative form, we need to evaluate the two operands (*g *and* h*) using interval arithmetic while we only need to evaluate one of them in the slope form. Note also that we could have defined the multiplication by* $[\mathtt{H}^{\vec{z}}](\vec{\mathrm{X}}) \times \mathsf{g}(\vec{\mathrm{X}}) + \mathsf{h}(\vec{z}) \times [\mathtt{G}^{\vec{z}}](\vec{\mathrm{X}})$ *(the division has also two forms) but the two possible forms of slope are over-approximations of* $\mathsf{f}(\vec{X})$*. Nevertheless, a possible way to choose between the two forms is to keep the form which gives the smallest approximation of* $\mathsf{f}(\vec{X})$*.*

In Figure 1, we give two graphical representations of interval slope expansion. For this purpose, we want to compute the image of x by the function $f(x) = x(1-x)+1$. We consider in Figure 1(a) that $x \in [-1, 1/2]$ and we get as a result that $f(x) \in [-1, 19/8]$ which is an over-approximation of the exact result $[-1, 5/4]$. The midpoint is $-1/4$ and the set of slopes is bounded by the interval $[0, 9/4]$. The dashed lines represent the linear approximation of the image. In Figure 1(b), we consider that $x \in [-1/2, 1/2]$ and the result is $f(x) \in [1/4, 7/4]$ which is still an over-approximation of the exact result $[1/4, 5/4]$. In that case, the midpoint is 0 and the set of slopes is bounded by the interval $[0, 3/2]$. Note that the smaller the interval the better the approximation is.

[6] In [20, Sect. 2], the authors went also into detail of the complexity of these operations.



(a) $x \in [-1, 1/2]$ (b) $x \in [-1/2, 1/2]$

Figure 1: Two examples of the interval expansion with slopes

Example 3 shows that we can encode with interval slopes the list of variables contributing in the result of an arithmetic expression. In particular, the vector composing the interval slope of the variable t represents the influence of the variables a, b and c on the value of t. For example, we know that a modification of the value of the variable a produce a modification of the result with the same order of the modification on a because the slope associated to a is $[1, 1]$. But a modification on the variable b by Δ_b will produce a modification on the t by $\Delta_b \times \mathrm{V}_c$ because the slope of b is equal to V_c.

Example 3. *Let $t = a + b \times c$, we want to compute the interval slope $[\mathtt{T}^z](\vec{\mathrm{X}})$ of t. We consider that $\mathcal{V}^{ind} = \{a, b, c\}$ and $\vec{\vec{\mathrm{X}}}$ is the interval vector of the values of these variables. We suppose that the interval slope expansion of a, b and c are $(z_a, [\mathtt{A}^{\vec{z}}](\vec{\mathrm{X}}) = \delta_1)$, $(z_b, [\mathtt{B}^{\vec{z}}](\vec{\mathrm{X}}) = \delta_2)$, and $(z_c, [\mathtt{C}^{\vec{z}}](\vec{\mathrm{X}}) = \delta_3)$ respectively. The interval value associated to c is* V_c i.e. $\mathrm{V}_c = z_c + [\mathtt{C}^z](\vec{\mathrm{X}})(\vec{\mathrm{X}} - \vec{z})$.

$$\begin{aligned}
[\mathtt{T}^{\vec{z}}](\vec{\mathrm{X}}) &= [\mathtt{A}^{\vec{z}}](\vec{\mathrm{X}}) + z_b[\mathtt{C}^{\vec{z}}](\vec{\mathrm{X}}) + [\mathtt{B}^{\vec{z}}](\vec{\mathrm{X}}) \left(z_c + [\mathtt{C}^{\vec{z}}](\vec{\mathrm{X}})(\vec{\mathrm{X}} - \vec{z}) \right) \\
&= ([1,1], 0, 0) + z_b \times (0, 0, [1,1]) + (0, [1,1], 0) \times \mathrm{V}_c \\
&= ([1,1], [1,1] \times \mathrm{V}_c, z_b \times [1,1]) \\
&= ([1,1], \mathrm{V}_c, [z_b, z_b])
\end{aligned}$$

As seen in Example 3, interval slopes represent relations between the inputs and the outputs of a function. By computing interval slopes, we build step by step the set of variables related to arithmetic expressions in programs. In static analysis, we can use this interval expansion to track the influence of the inputs of a program on its outputs. Hence the choice of the set $\mathcal{V}^{\mathrm{ind}}$ of independent variables is given by the set of the input variables of the program to analyse. Moreover, we can add in $\mathcal{V}^{\mathrm{ind}}$ all the other variables which may influence output.

3 Floating-Point Slopes

We present in this section our new abstract domain `FPS`. In Section 3.1, we adapt the computation rules of interval slopes to take into account floating-point arithmetic. Next in Section 3.2, we define an abstract semantics of arithmetic expressions over `FPS` values taking into account the behaviors of floating-point arithmetic. And in Section 3.3, we define the order structure of the `FPS` domain.

3.1 Floating-Point Version of Interval Slopes

The definition of interval slope expansion in Section 2.2 manipulates real numbers. In case of floating-point numbers, we have to take into account the round-off function and the rounding-errors.

We show in Proposition 1 that the range of a non-linear function f of floating-point numbers can be soundly over-approximated by a floating-point slope. The function f must respect the correct rounding, *i.e.* the property of Equation (2) must hold. In other words, the result of an operation over set of floating-point numbers is over-approximated by the result of the same operation over floating-point slopes by adding a small quantity depending on the relative rounding error unit μ and the absolute error σ.

Proposition 1. *Let $\mathsf{f} : D \subseteq \mathbb{R}^n \to \mathbb{R}$ be an arithmetic operation of the form $g \diamond h$ with $\diamond \in \{+, -, \times, \div\}$ or $\sqrt{\ }$, i.e. f respects the correct rounding. For all $\vec{\mathrm{X}} \subseteq D$ and $\vec{z} \in D$, we have:*

$$\mathrm{fl}\big(\mathsf{f}(\vec{\mathrm{X}})\big) \subseteq \mathsf{f}(\vec{z})\big(1 + [-\mu, \mu]\big) + \Big[-\frac{\sigma}{2}, \frac{\sigma}{2}\Big] + [\mathtt{F}^{\vec{z}}](\vec{\mathrm{X}})(X - z)\big(1 + [-\mu, \mu]\big) \ .$$

Proof.

$$\begin{aligned}
\mathrm{fl}\big(\mathsf{f}(\vec{\mathrm{X}})\big) &= \{\mathsf{f}(\vec{x})(1 + \varepsilon_x) + \bar{\varepsilon}_x : \vec{x} \in \vec{\mathrm{X}}\} && \text{by Eq. (2)}\\
&\subseteq \mathsf{f}(\vec{\mathrm{X}}) + \mathsf{f}(\vec{\mathrm{X}})\{\varepsilon_x : \vec{x} \in \vec{\mathrm{X}}\} + \{\bar{\varepsilon}_x : \vec{x} \in \vec{\mathrm{X}}\}\\
&\subseteq \big(\mathsf{f}(\vec{z}) + [\mathtt{F}^{\vec{z}}](\vec{\mathrm{X}})(\vec{\mathrm{X}} - \vec{z})\big) + \{\bar{\varepsilon}_x : \vec{x} \in \vec{\mathrm{X}}\} && \text{by Eq. (4)}\\
&\quad + \big(\mathsf{f}(\vec{z}) + [\mathtt{F}^{\vec{z}}](\vec{\mathrm{X}})(\vec{\mathrm{X}} - \vec{z})\big)\{\varepsilon_x : \vec{x} \in \vec{\mathrm{X}}\}\\
&\subseteq \mathsf{f}(\vec{z})\big(1 + \{\varepsilon_x : \vec{x} \in \vec{\mathrm{X}}\}\big) + \{\bar{\varepsilon}_x : \vec{x} \in \vec{\mathrm{X}}\}\\
&\quad + [\mathtt{F}^{\vec{z}}](\vec{\mathrm{X}})(\vec{\mathrm{X}} - \vec{z})\big(1 + \{\varepsilon_x : \vec{x} \in \vec{\mathrm{X}}\}\big)\\
&\subseteq \mathsf{f}(\vec{z})\big(1 + [-\mu, \mu]\big) + \Big[-\frac{\sigma}{2}, \frac{\sigma}{2}\Big] && |\varepsilon_x| \le \mu \text{ by Eq. (2)}\\
&\quad + [\mathtt{F}^{\vec{z}}](\vec{\mathrm{X}})(\vec{\mathrm{X}} - \vec{z})\big(1 + [-\mu, \mu]\big) && |\bar{\varepsilon}_x| \le \frac{1}{2}\sigma \text{ by Eq. (2)}
\end{aligned}$$

□ □

Remark 2. *As the floating-point version of slopes is based on μ and σ, we can represent the floating-point behaviors depending of the hardware. For example, extended precision[7] is represented using the values $\mu = 2^{-64}$ and $\sigma = 2^{-16446}$. Furthermore following [2], we can compute the result of a double rounding[8] with $\mu = (2^{11} + 2)2^{-64}$ and $\sigma = (2^{11} + 1)2^{-1086}$.*

Proposition 1 shows that we can compute the floating-point range of a function f, respecting the correct rounding, using interval slopes expansion. That is a set of floating-point values can is represented by a pair:

$$\Big([\mathsf{f}]\,(\vec{z})\big(1 + [-\mu, \mu]\,\big) + \Big[-\frac{\sigma}{2}, \frac{\sigma}{2}\Big], \quad [\mathtt{F}^{\vec{z}}](\vec{X})\big(1 + [-\mu, \mu]\big)\Big) \ .$$

The first element is a small interval rounding to the nearest around $\mathsf{f}(\vec{z})$ for which we have to take into account the possible rounding errors. The second element is the interval slopes which have to take account of relative errors. Note that this adaptation adds a very little overhead of computations compared to the definition of interval slopes by Krawczyk and Neumaier.

[7] In some hardware, *e.g.* Intel x87, floating-point numbers may be encoded with 80 bits in registers, *i.e.* the significand is 64 bits long.

[8] It may happen on hardware using extended precision. Results of computations are rounded in registers and they are rounded again, with a less precision, in memory.

3.2 Semantics of Arithmetic Operations

In this section, we define the abstract semantics of arithmetic operations over elements of floating-point slopes domain in order to mimic the behaviors of the floating-point arithmetic. We denote by $\mathbb{I}$ the set of intervals and by $\mathbb{S} = \mathbb{I} \times \mathbb{I}^{|\mathcal{V}^{\text{ind}}|}$ the set of slopes. An element s of $\mathbb{S}$ is represented by a pair $(\mathrm{M}, \vec{\mathrm{S}})$ where M is a floating-point interval and $\vec{\mathrm{S}}$ is a vector of floating-point intervals. We denote by $\langle \mathbb{I}, \sqsubseteq_{\mathbb{I}}, \perp_{\mathbb{I}}, \top_{\mathbb{I}}, \sqcup_{\mathbb{I}}, \sqcap_{\mathbb{I}} \rangle$ the lattice of intervals. First we define some auxiliary functions before presenting the semantics of arithmetic expressions over FPS.

The function ι defined in Equation (5) computes the interval value associated to a floating-point slopes $(\mathrm{M}, \vec{\mathrm{S}})$. We assume that the values of independent variables are kept in a separate interval vector $\vec{\mathrm{V}}_{\mathcal{V}^{\text{ind}}}$. The notation $\mathsf{mid}(\vec{\mathrm{V}}_{\mathcal{V}^{\text{ind}}})$ stands for the component-wise application of the function mid on all the components of the vector $\vec{\mathrm{V}}_{\mathcal{V}^{\text{ind}}}$. Note that $\cdot$ represents the scalar product.

$$\iota\big((\mathrm{M}, \vec{\mathrm{S}})\big) = \mathrm{M} + \vec{\mathrm{S}} \cdot \big(\vec{\mathrm{V}}_{\mathcal{V}^{\text{ind}}} - \mathsf{mid}(\vec{\mathrm{V}}_{\mathcal{V}^{\text{ind}}})\big) \tag{5}$$

The function κ defined in Equation (6) transforms an interval value $[a,b]^\ell$ associated to the ℓ-th independent variable into a floating-point slope.

$$\kappa\left([a,b]^\ell\right) = \big([m,m], \delta_\ell\big) \quad \text{with} \quad m = \mathsf{mid}([a,b]) \tag{6}$$

This function κ is used in two cases: *i)* To initialize all the independent variables at the beginning of an analysis. *ii)* In the meet operation, see Section 3.3.

We can detect overflows and generations of zero by using the function Φ defined in Equation (7). We have two kinds or rules: *total* rules when we are certain that a zero or an overflow occur and *partial* rules when a part of the set described by a floating-point slope generates a zero or an overflow. With the function ι we can determine for an element $(\mathrm{M}, \vec{\mathrm{S}}) \in \mathbb{S}$ if $(\mathrm{M}, \vec{\mathrm{S}})$ represents an overflow or a zero. Hence we represent the finite precision of the floating-point arithmetic. We denote by $\mathbf{p}_\infty$ and by $\mathbf{m}_\infty$ the interval vectors with all their components equal to $[+\infty, +\infty]$ and $[-\infty, -\infty]$ respectively. We recall that σ is the smallest denormalized and Σ is the largest floating-point numbers.

$$\Phi(\mathrm{M}, \vec{\mathrm{S}}) = \begin{cases} ([0,0], \mathbf{0}) & \text{if } \iota(\mathrm{M}, \vec{\mathrm{S}}) \sqsubseteq_{\mathbb{I}} [-\frac{\sigma}{2}, \frac{\sigma}{2}] \\ (\tilde{\mathrm{M}}, \mathbf{0} \,\dot{\sqcup}_{\mathbb{I}}\, S) & \text{if } \iota(\mathrm{M}, \vec{\mathrm{S}}) \sqcap_{\mathbb{I}}]-\frac{\sigma}{2}, \frac{\sigma}{2}[\neq \perp_{\mathbb{I}} \\ & \text{and } \tilde{\mathrm{M}} = \begin{cases} [0,0] & \text{if M } \sqsubseteq_{\mathbb{I}}]-\frac{\sigma}{2}, \frac{\sigma}{2}[\\ [0,0] \sqcup_{\mathbb{I}} \mathrm{M} & \text{otherwise} \end{cases} \\ ([+\infty, +\infty], \mathbf{p}_\infty) & \text{if } \iota(\mathrm{M}, \vec{\mathrm{S}}) \sqsubseteq_{\mathbb{I}}]\Sigma, +\infty] \\ (\tilde{\mathrm{M}}, \mathbf{p}_\infty \,\dot{\sqcup}_{\mathbb{I}}\, S) & \text{if } \iota(\mathrm{M}, \vec{\mathrm{S}}) \sqcap_{\mathbb{I}}]\Sigma, +\infty] \neq \perp_{\mathbb{I}} \\ & \text{and } \tilde{\mathrm{M}} = \begin{cases} [+\infty, +\infty] & \text{if M } \sqsubseteq_{\mathbb{I}}]\Sigma, +\infty] \\ [+\infty, +\infty] \sqcup_{\mathbb{I}} \mathrm{M} & \text{otherwise} \end{cases} \\ ([-\infty, -\infty], \mathbf{m}_\infty) & \text{if } \iota(\mathrm{M}, \vec{\mathrm{S}}) \sqsubseteq_{\mathbb{I}} [-\infty, -\Sigma[\\ (\tilde{\mathrm{M}}, \mathbf{m}_\infty \,\dot{\sqcup}_{\mathbb{I}}\, S) & \text{if } \iota(\mathrm{M}, \vec{\mathrm{S}}) \sqcap_{\mathbb{I}} [-\infty, -\Sigma[\neq \perp_{\mathbb{I}} \\ & \text{and } \tilde{\mathrm{M}} = \begin{cases} [-\infty, -\infty] & \text{if M } \sqsubseteq_{\mathbb{I}} [-\infty, -\Sigma[\\ [-\infty, -\infty] \sqcup_{\mathbb{I}} \mathrm{M} & \text{otherwise} \end{cases} \\ (\mathrm{M}, \vec{\mathrm{S}}) & \text{otherwise} \end{cases} \tag{7}$$

Equation (7) is an adaptation of the rule defined in Equation (1) to deal with FPS values. Furthermore, the abstract values $(+\infty, \mathbf{p}_\infty)$ and $(-\infty, \mathbf{m}_\infty)$ represent the special floating-point values $+\infty$ and $-\infty$ respectively. As in floating-point arithmetic, the values $(+\infty, \mathbf{p}_\infty)$ and $(-\infty, \mathbf{m}_\infty)$ are absorbing elements.

An interesting feature of interval slopes is that we can mimic the absorption phenomenon by setting to zero the interval slope of the absorbed operand. We define the function ρ for this purpose. Indeed, an abstract value $(\mathrm{M}, \vec{\mathrm{S}})$ already supports partial absorption as M is computed with a rounding to the nearest but $\vec{\mathrm{S}}$ have to be *reduced* to represent the absence of the influence of particular independent variables. The reduction of an abstract value $g = (\mathrm{M}_g, \vec{\mathrm{S}}_g)$ compared to an abstract value $h = (\mathrm{M}_h, \vec{\mathrm{S}}_h)$, denoted by $\rho(g \mid h)$, is defined in Equation (8).

$$\rho(g \mid h) = \begin{cases} ([0,0], \mathbf{0}) & \text{if } \iota(\mathrm{M}_g, \vec{\mathrm{S}}_g) \sqsubseteq_{\mathbb{I}} [\mu,\mu] \times \iota(\mathrm{M}_h, \vec{\mathrm{S}}_h) \\ (\tilde{\mathrm{M}}_g, \mathbf{0} \mathbin{\dot\sqcup_{\mathbb{I}}} \vec{\mathrm{S}}_g) & \text{if } \iota(\mathrm{M}_g, \vec{\mathrm{S}}_g) \sqcap_{\mathbb{I}} [\mu,\mu] \times \iota(\mathrm{M}_h, \vec{\mathrm{S}}_h) \neq \bot_{\mathbb{I}} \\ & \text{and } \tilde{\mathrm{M}}_g = \begin{cases} [0,0] & \text{if } \mathrm{M}_g \sqsubseteq_{\mathbb{I}} [\mu,\mu] \times \iota(\mathrm{M}_h, \vec{\mathrm{S}}_h) \\ [0,0] \sqcup_{\mathbb{I}} \mathrm{M}_g & \text{otherwise} \end{cases} \\ (\mathrm{M}_g, \vec{\mathrm{S}}_g) & \text{otherwise} \end{cases} \tag{8}$$

Equation (8) models the absorption phenomenon by explicitly setting to zero the values of a slope. As mentioned in Section 2.2, a slope shows which variables influence the computation of an arithmetic expression. But, absorption phenomena induce that an operand does not influence the result of an addition or a subtraction any more.

Using the functions Φ, ρ and ι, we inductively define on the structure of arithmetic expressions the abstract semantics $[\![.]\!]^{\sharp}_{\mathbb{S}}$ of floating-point slopes in Figure 2. We denote by $\mathrm{env}^{\sharp}$ an abstract environment which associates to each program variable a floating-point slope. For each arithmetic operation, we componentwisely combine the elements of the abstract operands $[\![g]\!]^{\sharp}_{\mathbb{S}}(\mathrm{env}^{\sharp}) = (\mathrm{M}_g, \vec{\mathrm{S}}_g)$ and $[\![h]\!]^{\sharp}_{\mathbb{S}}(\mathrm{env}^{\sharp}) = (\mathrm{M}_h, \vec{\mathrm{S}}_h)$. The element M is obtained using the interval arithmetic with rounding to the nearest. The element $\vec{\mathrm{S}}$ is computed using the definition of the slope arithmetic defined in Table 1. We take into account of the possible rounding errors in the result $(\mathrm{M}, \vec{\mathrm{S}})$ following Proposition 1. In case of addition and subtraction, according to the Equation (2), we do not consider absolute error $\frac{\sigma}{2}$ which is always zero. Moreover, in case of addition or subtraction, we handle the absorption phenomena using the function ρ, defined in Equation (8). Finally, we check if a zero or an overflow is generated by applying the function Φ defined in Equation (7).

$$[\![g \pm h]\!]^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) = \Phi\left((\tilde{\mathrm{M}}_g \pm \tilde{\mathrm{M}}_h)(1+[-\mu,\mu]),\quad \left(\tilde{\vec{\mathrm{S}}}_g \pm \tilde{\vec{\mathrm{S}}}_h\right)(1+[-\mu,\mu])\right)$$
$$\text{with} \quad (\tilde{\mathrm{M}}_g, \tilde{\vec{\mathrm{S}}}_g) = \rho(g \mid h) \text{ and } (\tilde{\mathrm{M}}_h, \tilde{\vec{\mathrm{S}}}_h) = \rho(h \mid g)$$

$$[\![g \times h]\!]^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) = \Phi\left(\mathrm{M},\quad (\vec{\mathrm{S}}_g \times \iota(\mathrm{M}_h, \vec{\mathrm{S}}_h) + \mathrm{M}_g \times \vec{\mathrm{S}}_h)(1+[-\mu,\mu])\right)$$
$$\text{with} \quad \mathrm{M} = (\mathrm{M}_g \times \mathrm{M}_h)(1+[-\mu,\mu]) + \left[\frac{\sigma}{2}, \frac{\sigma}{2}\right]$$

$$\left[\!\!\left[\frac{g}{h}\right]\!\!\right]^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) = \Phi\left(\mathrm{M},\quad \frac{\vec{\mathrm{S}}_g - \vec{\mathrm{S}}_h \frac{\mathrm{M}_g}{\mathrm{M}_h}}{\iota(\mathrm{M}_h, \vec{\mathrm{S}}_h)}(1+[-\mu,\mu])\right)$$
$$\text{with} \quad \mathrm{M} = \frac{\mathrm{M}_g}{\mathrm{M}_h}(1+[-\mu,\mu]) + \left[\frac{\sigma}{2}, \frac{\sigma}{2}\right],$$
$$0 \notin \iota(\mathrm{M}_h, \vec{\mathrm{S}}_h) \text{ and } 0 \notin \mathrm{M}_h$$

$$[\![\sqrt{g}]\!]^{\sharp}_{\mathbb{S}}(\theta^{\sharp}) = \Phi\left(\mathrm{M},\quad \left(\frac{\vec{\mathrm{S}}_g}{\sqrt{\mathrm{M}_g} + \sqrt{\iota(\mathrm{M}_g, \vec{\mathrm{S}}_g)}}\right)(1+[-\mu,\mu])\right)$$
$$\text{with} \quad \mathrm{M} = \left(\sqrt{\mathrm{M}_g}\,(1+[-\mu,\mu])\right) + \left[-\frac{\sigma}{2}, \frac{\sigma}{2}\right],$$
$$\mathrm{M}_g \sqcap_{\mathbb{I}} [-\infty, 0] = \bot_{\mathbb{I}} \text{ and } \iota(\mathrm{M}_g, \vec{\mathrm{S}}_g) \sqcap_{\mathbb{I}} [-\infty, 0] = \bot_{\mathbb{I}}$$

Figure 2: Abstract semantics of arithmetic expressions on floating-point slopes

Remark 3. *The functions Φ and ρ make the arithmetic operations on floating-point slopes non associative and non distributive as in floating-point arithmetic.*

3.3 Order Structure

In this section, we define the order structure of the set $\mathbb{S}$ of floating-point slopes. In particular, this structure is based on the lattice of intervals. We recall that the set of slopes $\mathbb{S} = \mathbb{I} \times \mathbb{I}^{|\mathcal{V}^{\text{ind}}|}$ and an element s of $\mathbb{S}$ is a pair $(\mathrm{M}, \vec{\mathrm{S}})$.

We define a partial order, the join and the meet operations between elements of $\mathbb{S}$. All these operations are defined as a component-wise application of the associated operations of the interval domain except the meet operation which needs extra care. We denote by $\dot{\sqsubseteq}_{\mathbb{I}}$ the component-wise application of the interval order. We can define a partial order $\sqsubseteq_{\mathbb{S}}$ between elements of $\mathbb{S}$ with:

$$\forall(\mathrm{M}_g, \vec{\mathrm{S}}_g), (\mathrm{M}_h, \vec{\mathrm{S}}_h) \in \mathbb{S}, \left(\mathrm{M}_g, \vec{\mathrm{S}}_g\right) \sqsubseteq_{\mathbb{S}} \left(\mathrm{M}_h, \vec{\mathrm{S}}_h\right) \Leftrightarrow \mathrm{M}_g \sqsubseteq_{\mathbb{I}} \mathrm{M}_h \wedge \vec{\mathrm{S}}_g \dot{\sqsubseteq}_{\mathbb{I}} \vec{\mathrm{S}}_h . \quad (9)$$

The join operation $\sqcup_{\mathbb{S}}$ over floating-point slopes is defined in Equation (10). We denote by $\dot{\sqcup}_{\mathbb{I}}$ the component-wise application of the operation $\sqcup_{\mathbb{I}}$.

$$\forall(\mathrm{M}_g, \vec{\mathrm{S}}_g), (\mathrm{M}_h, \vec{\mathrm{S}}_h) \in \mathbb{S}, \quad \left(\mathrm{M}_g, \vec{\mathrm{S}}_g\right) \sqcup_{\mathbb{S}} \left(\mathrm{M}_h, \vec{\mathrm{S}}_h\right) = \left(\mathrm{M}, S\right)$$
$$\text{with} \quad \mathrm{M} = \mathrm{M}_g \sqcup_{\mathbb{I}} \mathrm{M}_h \quad \text{and} \quad S = \vec{\mathrm{S}}_g \dot{\sqcup}_{\mathbb{I}} \vec{\mathrm{S}}_h \quad (10)$$

There is no direct way to define the greatest lower bound of two elements of $\mathbb{S}$. Indeed, two abstract values may represent the same concrete value but without being comparable. Hence we only have a join-semilattice structure. The meet operation $\sqcap_{\mathbb{S}}$ over floating-point slopes is defined in Equation (11). It may require a conversion into interval value. We consider that the result of the meet operation introduces a new independent variable at index ℓ. We denote by $\sqsubset_{\mathbb{I}}$ the strict comparison of intervals and by $\perp_{\mathbb{S}}$ the least element of $\mathbb{S}$.

$$\forall(\mathrm{M}_g, \vec{\mathrm{S}}_g), (\mathrm{M}_h, \vec{\mathrm{S}}_h) \in \mathbb{S}, \quad \left(\mathrm{M}_g, \vec{\mathrm{S}}_g\right) \sqcap_{\mathbb{S}} \left(\mathrm{M}_h, \vec{\mathrm{S}}_h\right) =$$
$$\begin{cases} \perp_{\mathbb{S}} & \text{if } \iota(\mathrm{M}_g, \vec{\mathrm{S}}_g) \sqcap_{\mathbb{I}} \iota(\mathrm{M}_h, \vec{\mathrm{S}}_h) = \perp_{\mathbb{I}} \\ (\mathrm{M}_g, \vec{\mathrm{S}}_g) & \text{if } \iota(\mathrm{M}_g, \vec{\mathrm{S}}_g) \sqsubset_{\mathbb{I}} \iota(\mathrm{M}_h, \vec{\mathrm{S}}_h) \\ (\mathrm{M}_h, \vec{\mathrm{S}}_h) & \text{if } \iota(\mathrm{M}_h, \vec{\mathrm{S}}_h) \sqsubset_{\mathbb{I}} \iota(\mathrm{M}_g, \vec{\mathrm{S}}_g) \\ \kappa\left(\iota(\mathrm{M}_h, \vec{\mathrm{S}}_h) \sqcap_{\mathbb{I}}^{\ell} \iota(\mathrm{M}_g, \vec{\mathrm{S}}_g)\right) & \text{otherwise} \end{cases} \quad (11)$$

Note on the Widening Operator.

In order to enforce the convergence of the fixpoint computation, we can define a widening operation $\nabla_{\mathbb{S}}$ over floating-point slopes values. An advantage of our domain is that we can straightforwardly use the widening operations defined for the interval domain denoted by $\nabla_{\mathbb{I}}$. We define the operator $\nabla_{\mathbb{S}}$ in Equation (12) using the widening operator between intervals. The notation $\dot{\nabla}_{\mathbb{I}}$ represents the component-wise application of $\nabla_{\mathbb{I}}$ between the components of the interval slopes vector.

$$\forall(\mathrm{M}_g, \vec{\mathrm{S}}_g), (\mathrm{M}_h, \vec{\mathrm{S}}_h) \in \mathbb{S}, \quad \left(\mathrm{M}_g, \vec{\mathrm{S}}_g\right) \nabla_{\mathbb{S}} \left(\mathrm{M}_h, \vec{\mathrm{S}}_h\right) = \left(\mathrm{M}, \vec{\mathrm{S}}\right)$$
$$\text{with} \quad \mathrm{M} = \mathrm{M}_g \nabla_{\mathbb{I}} \mathrm{M}_h \quad \text{and} \quad \vec{\mathrm{S}} = \vec{\mathrm{S}}_g \dot{\nabla}_{\mathbb{I}} \vec{\mathrm{S}}_h \quad (12)$$

4 Analysis of Floating-Point Programs

The goal of the static analysis of floating-point programs using the floating-point slopes domain is to give for each control point and for each variable an over-approximation given by FPS of the reachable set of floating-point numbers. An abstract environment $\text{env}^\sharp$ associates to each variable $v \in \mathcal{V}$ a value of $\mathbb{S}$. The set $\mathcal{V}$ is made of the sets $\mathcal{V}^{\text{ind}}$ and $\mathcal{V}^{\text{dep}}$ of independent and dependent variables.

The semantics of an assignment $[\![v := e]\!]^\sharp$ in the abstract environment $\text{env}^\sharp$ is the update of the value associated to v with the result of the evaluation of the arithmetic expression e using the arithmetic operations over FPS given in Figure 2. As the FPS domain is related to the interval domain we can straightforwardly use the semantics of tests given in [15] to refine the value of variables. Note that the semantics of tests is related to the meet operation defined in Equation (11) which may conserve some relations between variables.

We define in Equation (13) the concretization function $\gamma_{\mathbb{S}}$ between the join-semilattice $\langle \mathcal{V} \to \mathbb{S}, \dot{\sqsubseteq}_{\mathbb{S}} \rangle$, with $\dot{\sqsubseteq}_{\mathbb{S}}$ the point-wise lifting comparison, and the complete lattice $\langle \wp(\mathcal{V} \to \mathbb{F}), \subseteq \rangle$.

$$\gamma_{\mathbb{S}}\big(v \mapsto \big(\mathrm{M}, \vec{\mathrm{S}}\big)\big) = \bigcup_{\vec{u} \in \vec{\mathrm{V}}_{\mathcal{V}^{\mathrm{ind}}}} \left\{ v \mapsto i \in \mathrm{I} : \mathrm{I} = \mathrm{M} + \vec{\mathrm{S}} \cdot \left(\vec{u} - \mathsf{mid}\big(\vec{\mathrm{V}}_{\mathcal{V}^{\mathrm{ind}}}\big)\right) \right\} \tag{13}$$

In Theorem 1, we state the soundness of the floating-point analysis using FPS domain with respect to the concrete floating-point semantics. The later is based on the concrete semantics of floating-point expressions $[\![e]\!]$, see [24] for its definition.

Theorem 1. *If the set of concrete environments env is contained in the abstract environment $env^\sharp$ then we have for all instruction i representing either an assignment or a test:*

$$[\![i]\!](env) \subseteq \gamma_{\mathbb{S}}\left([\![i]\!]^\sharp(env^\sharp) \right) \quad .$$

5 Case Studies

In this section, we present experimental results of the static analysis of numerical programs using our floating-point slope domain. We based our examples on Matlab/Simulink models which are block-diagrams. We present as examples a second order linear filter and a square root computation with a Newton method.

We first give a quick view of Matlab/Simulink models. In a block-diagram, each node represents an operation and each wire represents a value evolving during time. We consider a few operations such that arithmetic operations, gain operation that is multiplication by a constant, conditional statement (called *switch*[9] in Simulink), and *unit delay* block represented by $\frac{1}{\mathrm{z}}$ which acts as a memory. We can hence write discrete-time models thanks to finite difference equations, see [3] for further details.

The semantics of Simulink models is based on finite-time execution. In other words, a Simulink model is implicitly embedded in a *simulation loop* modelling the temporal evolution starting from $t = 0$ to a given final time t_{end}. The body of this loop follows three steps: *i)* evaluating the inputs, *ii)* computing the outputs, *iii)* updating the state variables *i.e.* values of the unit delay blocks. The static analysis of Simulink models transforms the simulation loop into a fixpoint computation. In its simple form, see [3] for further details, we add an extra time instant to collect all the behaviors from t_{end} to $t = +\infty$.

Linear Filter.

We applied the floating-point slope domain on a second order linear filter defined by: $y_n = x_n + 0.7x_{n-1} + x_{n-2} + 1.2y_{n-1} - 0.7y_{n-2}$.

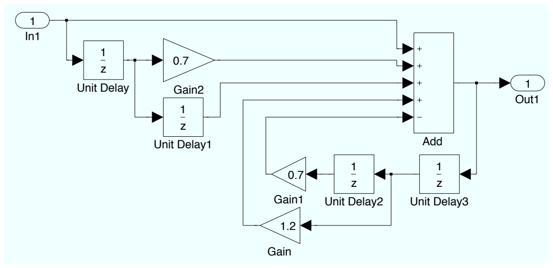
The block-diagrams of this filter is given in Figure 3(a). We consider a simulation time of 25 seconds that is we unfold the simulation loop 25 times before making unions. The input belongs into the interval $[0.71, 1.35]$. The output of the filter is given in Figure 3(b). We consider, in this example, that $\mathcal{V}^{\mathrm{ind}}$ contains the input and the four unit delay blocks that is there are five independent variables. The gray area represents all the possible trajectories of the output corresponding of the set of inputs. Hence we can bound the output, without using the widening operator, by the interval $[0.7099, 9.8269]$.

Newton Method.

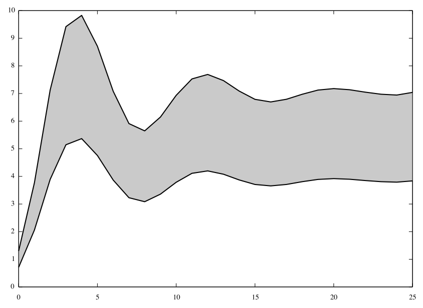
We applied our domain on a Newton algorithm which computes the square root of a number a using the following iterative sequence: $x_{n+1} = \frac{x_n}{2} + \frac{a}{2x_n}$.

We want to compute x_5 that is we consider the result of the Newton method after five iterations. The Simulink model is given in Figure 4(a) and in Figure 4(b), we give the model associated to one iteration of

[9]This operation is equivalent to the conditional expression: if $p_c(e_0)$ then e_1 else e_2. The predicate p_c has the form $e_0 \diamond c$ where c is a given constant and $\diamond \in \{\geq, >, \neq\}$.

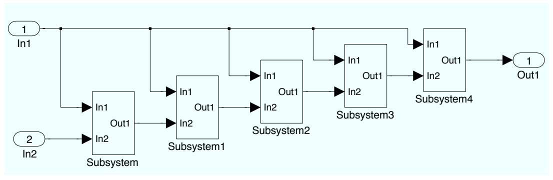


(a) Simulink model

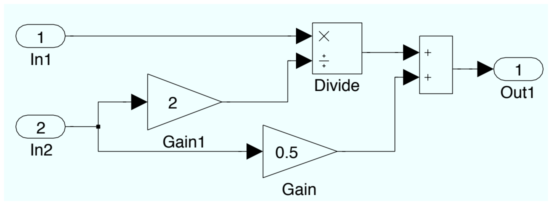


(b) Temporal evolution of the output

Figure 3: Second order linear filter



(a) Main model



(b) Content of a subsystem

Figure 4: Simulink model of the square root computation

the algorithm. In this case, the set $\mathcal{V}^{\text{ind}}$ is only made of one element. For the interval input $[4, 8]$ with the initial value equals to 2, we have the result $[1.8547, 3.0442]$.

6 Related Work

Numerical domains have been intensively studied. A large part of numerical domains concern the polyhedral representation of sets. For example, we have the domain of polyhedron [10] and the variants [32, 25, 31, 29, 8, 22, 21, 6, 7]. We also have the numerical domains based on affine relations between variables [19, 12] or the domain of linear congruences [16]. In general, all these domains are based on arithmetic with "good" properties such that rational numbers or real numbers. A notable exception is the floating-point versions of the octagon domain [24] and of the domain of polyhedron [5]. These domains give a sound over-approximation of the floating-point behaviors but they are not empowered to model the behaviors of floating-point arithmetic as we do.

Our FPS domain is more general than numerical abstract domains made for a special purpose. For example, we have the domain for linear filters [11] or for the numerical precision [14] which provide excellent results. Nevertheless as we showed in Section 5, we can apply this domain in various situations without losing too much precision.

7 Conclusion

We presented a new partially relational abstract numerical domain called FPS dedicated to floating-point variables. It is based on Krawczyk and Neumaier's work [20] on interval expansion of rational function using interval slopes. This domain is able to mimic the behaviors of the floating-point arithmetic such that the *absorption* phenomenon. We also presented experimental results showing the practical use of this domain in various contexts.

We want to pursue the work on the FPS domain by refining the the meet operation in order to keep relations between variables. Moreover we would like to model more closely the behaviors of floating point arithmetic, for example by taking into account the hardware instructions [26, Sect. 3].

As an other future work, we want to apply FPS domain for the analyses of the numerical precision by combining the FPS domain and domains defined in [23, 4]. An interesting direction should be to make an analysis of the numerical precision by comparing results of the FPS domain and results coming from the other numerical domain which bound the exact mathematical behaviors such that [5]. Hence we can avoid the manipulation of complex abstract values to represent rounding errors such as in [23, 14, 4].

Acknowledgements.

The author deeply thanks O. Bouissou, S. Graillat, T. Hilaire, D. Massé and M. Martel for their useful comments on the earlier versions of this article. He is also very grateful to anonymous referees who helped improving this work.

References

[1] C. H. Bischof, P. D. Hovland, and B. Norris. Implementation of automatic differentiation tools. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107. ACM, 2002.

[2] S. Boldo and T.M.T. Nguyen. Hardware-independant proofs of numerical programs. In *NASA Formal Methods Symposium*, 2010.

[3] A. Chapoutot and M. Martel. Abstract simulation: a static analysis of Simulink models. In *International Conference on Embedded Systems and Software*, pages 83–92. IEEE Press, 2009.

[4] Alexandre Chapoutot and Matthieu Martel. Automatic differentiation and Taylor forms in static analysis of numerical programs. *Technique et Science Informatiques*, 28(4):503–531, 2009. in French.

[5] Liqian Chen, Antoine Miné, and Cousot Patrick. A sound floating-point polyhedra abstract domain. In *Asian Symposium on Programming Languages and Systems*, volume 5356 of *LNCS*, pages 3–18. Springer, 2008.

[6] Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. Interval polyhedra: an abstract domain to infer interval linear relationships. In *Static Analysis Symposium*, volume 5673 of *LNCS*, pages 309–325. Springer, 2009.

[7] Liqian Chen, Antoine Miné, Ji Wang, and Patrick Cousot. An abstract domain to discover interval linear equalities. In *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *LNCS*, pages 112–128. Springer, 2010.

[8] Robert Clarisó and Jordi Cortadella. The Octahedron abstract domain. *Science Computer Programming*, 64(1):115–139, 2007.

[9] P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252. ACM, 1977.

[10] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages*, pages 84–97. ACM, 1978.

[11] J. Férêt. Static analysis of digital filter. In *European Symposium on Programming*, volume 2986 of *LNCS*, pages 33–48. Springer, 2004.

[12] K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain Taylor1+. In *Computer Aided Verification*, pages 627–633, 2009.

[13] E. Goubault. Static analyses of floating-point operations. In *Static Analysis Symposium*, volume 2126 of *LNCS*, pages 234–259. Springer, 2001.

[14] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Static Analysis Symposium*, volume 4134 of *LNCS*, pages 18–34. Springer, 2006.

[15] P. Granger. Improving the results of static analyses programs by local decreasing iteration. In *Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *LNCS*, pages 68–79. Springer, 1992.

[16] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT Vol.1*, volume 493 of *LNCS*, pages 169–192. Springer, 1991.

[17] N.J. Higham. *Accuracy and stability of numerical algorithms.* Society for Industrial and Applied Mathematics, 2nd edition, 2002.

[18] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic.* Institute of Electrical, and Electronic Engineers, 2008.

[19] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[20] R. Krawczyk and A. Neumaier. Interval slopes for rational functions and associated centered forms. *SIAM Journal on Numerical Analysis*, 22(3):604–616, 1985.

[21] Vincent Laviron and Francesco Logozzo. Subpolyhedra: a (more) scalable approach to infer linear inequalities. In *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 229–244, 2009.

[22] F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *Symposium on Applied Computing*, pages 184–188. ACM, 2008.

[23] M. Martel. Semantics of roundoff error propagation in finite precision computations. *Higher Order and Symbolic Computation*, 19(1):7–30, 2004.

[24] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *European Symposium on Programming*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[25] A. Miné. The Octagon abstract domain. *Journal of Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[26] D. Monniaux. Compositional analysis of floating-point linear numerical filters. In *Computer-Aided Verification*, volume 3576 of *LNCS*, pages 199–212. Springer, 2005.

[27] R. Moore. *Interval analysis.* Prentice Hall, 1966.

[28] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of floating-point arithmetic.* Birkhauser Boston, 2009.

[29] Mathias Péron and Nicolas Halbwachs. An abstract domain extending difference-bound matrices with disequality constraints. In *Verification, Model Checking and Abstract Interpretation*, volume 4349 of *LNCS*, pages 268–282. Springer, 2007.

[30] S.M. Rump. Expansion and estimation of the range of nonlinear functions. *Mathematics of Computation*, 65(216):1503–1512, 1996.

[31] Sriram Sankaranarayanan, Michael Colon, Henny Sipma, and Zohar Manna. Efficient strongly relational polyhedral analysis. In *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *LNCS*, pages 111–125. Springer Verlag, 2006.

[32] Axel Simon, Andy King, and Jacob Howe. Two variables per linear inequality as an abstract domain. In *Logic Based Program Synthesis and Transformation*, volume 2664 of *LNCS*, pages 71–89, 2003.